Area of Simulation

Mechanism and Architecture for Multi-Avatar Virtual Environments

Shen, Siqi; Iosup, Alexandru; Epema, Dick; Hu, Shun-Yun

DOI
10.1145/2764463

Publication date
2015

Document Version
Accepted author manuscript

Published in
ACM Transactions on Multimedia Computing, Communications, and Applications

Area of Simulation: Mechanism and Architecture for Multi-Avatar Virtual Environments

SIQI SHEN, ALEXANDRU IOSUP, DICK EPEMA, Delft University of Technology, The Netherlands
SHUN-YUN HU, Academia Sinica, Taiwan, R.O.C.

Although Multi-Avatar Distributed Virtual Environments (MAVEs) such as Real-Time Strategy (RTS) games entertain daily hundreds of millions of online players, their current designs do not scale. For example, even popular RTS games such as the StarCraft series support in a single game instance only up to 16 players and only a few hundreds of avatars loosely controlled by these players, which is a consequence of the Event-Based Lockstep Simulation (EBLS) scalability mechanism they employ. Through empirical analysis, we show that a single Area of Interest (AoI), which is a scalability mechanism that is sufficient for single-avatar virtual environments (such as Role-Playing Games), also cannot meet the scalability demands of MAVEs. To enable scalable MAVEs, in this work we propose Area of Simulation (AoS), a new scalability mechanism, which combines and extends the mechanisms of AoI and EBLS. Unlike traditional AoI approaches, which employ only update-based operational models, our AoS mechanism uses both event-based and update-based operational models to manage not single, but *multiple* areas of interest. Unlike EBLS, which is traditionally used to synchronize the entire virtual world, our AoS mechanism synchronizes only selected areas of the virtual world. We further design an AoS-based architecture, which is able to use both our AoS and traditional AoI mechanisms simultaneously, dynamically trading-off consistency guarantees for scalability. We implement and deploy this architecture and we demonstrate that it can operate with an order of magnitude more avatars and a larger virtual world without exceeding the resource capacity of players' computers.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Client/server*; C.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems—*Artificial, augmented, and virtual realities*

Additional Key Words and Phrases: Distributed Virtual Environments, Area of Interest, Real Time Strategy Games

1. INTRODUCTION

Multi-Avatar Virtual Environments (MAVEs), such as Real-Time Strategy (RTS) [Buro and Churchill 2012] games, have a large market, with millions of users [ESA 2012]. Contrary to the trend of Internet-based applications of allowing massive numbers of users to interact, the current generation of MAVEs design technology is not scalable. As a typical example, the StarCraft series limits the number of concurrent players in any gaming instance to 16; although hundreds of thousands of instances may run concurrently, they are essentially not communicating. This significant scalability limit stems from the difficulty of managing more than the hundreds of avatars that even this small number of players control in each game instance. Much previous work has focused on the scalability of Distributed Virtual Environments (DVEs) [Gilmore and Engelbrecht 2012; Liu et al. 2012; Yahyavi and Kemme 2013], leading to mechanisms such as Event-Based Lockstep Simulation (EBLS) [Terrano and Bettner 2001][Fiedler 2010] used in RTS games and military simulations, and the Area of Interest (AoI) [Ahmed and Shirmohammadi 2009] used in Single-Avatar Virtual Environments (SAVEs). However, as we show in this work, these mechanisms cannot be used to scale current MAVEs far beyond their current limits. To address the problem of scaling MAVEs, we propose the *Area of Simulation* scalability mechanism, we design an architecture around it, and we implement and deploy this architecture to demonstrate its scalability.

In MAVEs, users can have and control simultaneously multiple avatars which are their virtual world representations. We focus in this work on an important type of

MAVEs: RTS games, such as Blizzard's StarCraft and Microsoft's Age of Empires series, are essentially Internet-based real-time world simulations in which players control avatars to gather resources, to construct buildings, to train combat avatars, to explore unknown territories, to trade, and to conquer.

Two main problems prevent MAVEs from scaling. First, the resource capabilities of individual players may be exceeded: the bandwidth can become insufficient for transmitting messages, the computers of the players can become overloaded in trying to update the local copies of the game world status, etc. Second, MAVEs require strong consistency among the players for important areas of the virtual world: deciding which vehicle to move, where to build an important warehouse are decisions of precise location. Providing a consistent view to all players is challenging. When the scalability requirements are not met, the consequences for the game operators can be significant: players may quit en masse.

Two of the most commonly used scalability mechanisms: EBLS and AoI, do not work for MAVEs. EBLS [Gilmore and Engelbrecht 2012; Terrano and Bettner 2001; Fiedler 2010], the predominant *event-based* operational model for RTS games, uses lockstep simulation [Baughman and Levine 2001] to ensure a globally consistent execution order of events. EBLS trades off computation for bandwidth, by transmitting only events and by having every player recompute the state from the received events. EBLS consumes lots of computational power, and it cannot scale to hundreds of players on commodity computers, as we have previously shown in our evaluation of RTS games [Shen et al. 2011]. AoI uses an *update-based* operational model in which clients do not perform simulation of game states, but receive state-updates for objects in close in-game proximity. AoI-based approaches can scale to hundreds of concurrent avatars for *single* AoI. In this work, we analyze a large number of game traces and show that single-AoI approaches are not suitable for RTS games, which exhibit *multiple*, often-changing AoIs.

We propose, in this work, the *Area of Simulation (AoS)* scalability mechanism, which combines and extends the EBLS and AoI mechanisms. The AoS mechanism allows different areas of the virtual world to employ different operation models, from *event-based* to *update-based*, depending on the recent interest shown by the player. The AoS mechanism is the first mechanism to combine the event-based and the update-based operational models for managing the areas that a player is interested in.

We further design a system architecture for MAVEs with as its main feature the support of multiple, dynamic AoIs managed using the AoS mechanism. This architecture also includes two message dissemination mechanisms to reduce bandwidth consumption. We demonstrate the viability of our architecture through realistic simulations and through real-world experiments with a prototype game. By implementing and deploying a working system, we show that for a prototype yet realistic game, our architecture enables an order of magnitude more users than the state-of-the-art while satisfying the overall requirements of MAVEs.

In summary, our main contribution is five-fold:

(1) We show that most MAVEs users have multiple areas of interest, which can change often during gameplay. Thus, traditional approaches with a single AoI work poorly (Section 3);
(2) We propose a new scalability mechanism for MAVEs, the Area of Simulation (Section 4).
(3) We propose a system architecture for MAVEs based on Area of Simulation that can scale to hundreds of concurrent players with tens of thousands of avatars (Section 5).

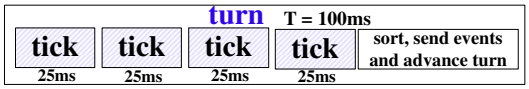

Fig. 1. Turn and tick.

(4) We implement this architecture and evaluate the architecture in simulation (Section 6) and with a real-world prototype RTS game (Section 7).
(5) We compare our work with a large body of related approaches, both quantitatively (Section 6 and 7) and qualitatively (Section 8).

2. CHARACTERISTICS AND SYSTEM MODEL

2.1. Characteristics and requirements of MAVEs

MAVEs such as RTS games have unique characteristics and requirements among DVEs [Claypool 2005]. Unlike other DVEs, such as First-Person Shooter (FPS) and Role-Playing Games (RPG), in which the player controls one avatar and may encounter at any time at most a few tens of other avatars (often not human-controlled characters), in RTS games the players often need to control many tens or even hundreds of avatars and in-game buildings, etc.

The control in RTS games combines long-term strategic decisions, including macro-management of resources such as buildings and large groups of avatars; short-term strategic decisions, including management of small groups of avatars; and quick tactical decisions, including micro-management of individual units. Usually, players expect the latencies not to exceed several seconds until the commands they issue are executed, or even less [Claypool 2005; Miller 2011]. Moreover, even if individual commands take long to be executed, the overall responsiveness of the game should not be compromised: players expect to see their game visuals updated at a rate of over 24 frames per second [Gregory 2009].

2.2. A System Model for MAVEs

In MAVEs, each user can have multiple avatars and in-game buildings. Each avatar has a pre-defined speed and range of vision. Typically each MAVE user has a base to produce/train the user's avatars.

Many RTS games, such as StarCraft, Age of Empires, 0 A.D, OpenTTD, and Zero-k, are EBLS-based systems [Fiedler 2010; Terrano and Bettner 2001], for which events can be triggered not only by user input (commands) but also by the (discrete) passage of time. Events are spatial and temporal, that is, they have a well-defined area and duration of effect. In this work, we use the terms "command" and "event" interchangeably.

In EBLS, the virtual world simulation is temporally divided into multiple simulation *turns*. Each turn has a pre-defined real world length T ms; after T ms, a turn is cutoff and a new turn is started. A *turn* is further divided into multiple simulation *ticks*. During each tick, the virtual world will perform simulation logic and render virtual-world objects' updates. In Figure 1, a turn's duration is set to $100\,ms$, and the turn is divided into 4 ticks, with each tick's duration equal to $25\,ms$. During a turn, each client will send events to a server. At the end of a turn, the clients send turn advance messages to the server. Upon receiving the turn advance messages, the server will sort all the events received and send the sorted events to all clients, for execution. According to [Terrano and Bettner 2001] and our own experience, the time spent for sorting and sending (non-blocking) events is negligible. A tick can be rendered using one or multiple frames; in this article, a tick is equivalent to one frame and we use the terms "frame" and "tick" interchangeably.

3. PROBLEM STATEMENT

In this section, we identify and discuss, in turn, three main challenges in fulfilling the per-command and overall latency requirements of MAVEs (Section 3.1). We also show that the traditional AoI mechanism, which is widely used to scale DVEs, is not efficient for MAVEs (Section 3.2). Thus, a new scalability mechanism and an accompanying architecture are needed to scale MAVEs.

3.1. Challenges in fulfilling MAVEs latency requirements

Resource challenge: Scaling MAVEs under tight latency requirements can be limited by the lack of sufficient computational and networking resources. Although the average upload-bandwidth required by RTS games is 2-8 KB/s for 8 players, it increases quadratically with the number of players [Claypool 2005], and for 100 players it can easily exceed 1 MB/s. We have also shown in our previous evaluation of RTS games [Shen et al. 2011] that, as the number of players increases, the computational resources required to update the game world can exceed the local computing power of modern commodity computers.

Game-design scalability challenge: We have also shown [Shen et al. 2011] that to deliver good gameplay experience when the number of players increases, a proportional increase in the size of the virtual world needs to occur, making the simulation of the virtual world even more computationally demanding than in today's commercial games.

Consistency challenge: Current and future RTS games require good consistency among players, especially for important areas of the game map (e.g, places of interest). It has been noted [Claypool 2005] that RTS games do not require location consistency as accurate as for FPS and RPG games, where the accuracy may make the difference between a player dying or living in the virtual world. However, avatar micromanagement, which has recently become very popular due to the release of games such as StarCraft and to the growth of global competition networks [Miller 2011], requires game-state consistency on-par with FPS and RPG games among the players simultaneously moving avatars in tight areas. For example, a trooper may be saved from disappearing by moving it in time just outside the fire range of an opponent's tank.

3.2. Presence of Areas of Interest in MAVEs

In this section, we show that *the AoI mechanism cannot support MAVEs well*. The AoI mechanism, adopted by many DVEs, exploits the interest shown by users to specific avatars or map areas, to reduce the traffic needed for progressing to the next simulation tick. However, previous approaches use only a *single* AoI per user, the location of which is defined as the area surrounding the virtual world location around the user's *single* avatar[1]. To study the potential use of AoI in MAVEs, we analyze the real use of StarCraft II (SC), one of the most popular RTS games, as a representative MAVE. Through the analysis of about 6,000 logs of SC matches, we show that most MAVEs users have each not a single but *multiple* AoIs in game, and that players switch quickly among their set of AoIs.

We collect 5,796 replays of SC from `sc2rep.net`, a popular repository of community-rated game replays. The replays have been created and uploaded to the website, for review by other players, by over 1,000 users. The replays are played between July 2010 and November 2010, and the average duration of the replays is about 13 minutes. We

[1]Some SAVE games such as World of Warcraft, allow players to level-up multiple avatars, but the avatars cannot be controlled simultaneously in the same game instance

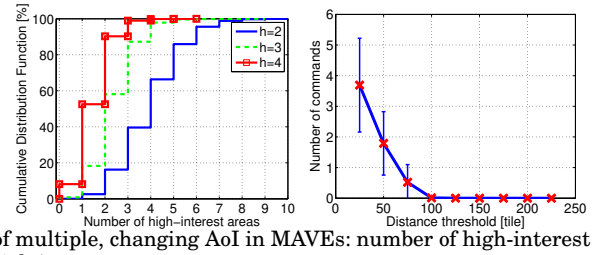

Fig. 2. The presence of multiple, changing AoI in MAVEs: number of high-interest areas per user (left) and dynamics of interest (right).

use the publicly available tool `SC2Gear` to extract from each replay the complete set of timestamped, location-aware commands.

The size of SC maps ranges from 64×64 to $256 \times 256\, tiles$. The speed of the fastest moving unit is about $7.5\, tiles/s$ and the broadest range of vision of in-game units is $14\, tiles$. The actual size of the map area that is viewable on-screen depends on the aspect ratio of the user's monitor, but we conservatively estimate that each screen can display map areas of about $25 \times 25\, tiles$. Thus, we divide for each replay the map into areas of $25 \times 25\, tiles$ and count the commands issued in each area. Because we are looking for areas of much higher than average interest, which correspond to the intuition behind AoI, we define a *high-interest area* as the area for which the number of issued commands is h times higher than the average number of commands issued per area.

Most users have multiple AoIs. Figure 2 (left) depicts the distribution of the number of high-interest areas per player when setting $h = 2, 3, 4$. As the figure shows, for $h = 2$, only less than 5% of the players have one high-interest area and about 90% of the players have more than two high-interest areas. The maximal number of high-interest areas of a player is 16. For $h = 3$, only 10% of the players have one high-interest areas and about 80% of the players have more than two high-interest areas. For $h = 4$, over 40% of the users have 2 or more high-interest areas. Overall, we conclude that most of the players have 2–6 high-interest areas. This can be explained by observing that advanced players employ a mix of macro- and micro-management (see Section 2.1) in different areas of the game map.

Users switch among their AoIs in the virtual world, often with high frequencies. We look at the distance between the virtual world location of commands issued over short periods of time. For each replay, we split the duration in a series of 10-second time periods, and analyze the commands issued in each period. We consider a distance threshold x, in turn, from 25 (the screen size) to 225 (the maximum map size minus the screen size), in increments of 25. For each distance threshold and each period, we count the number of commands issued further than the threshold from the location of the first command in the period; such a command would require an AoI switch. Figure 2 (right) depicts the mean command-counts for various distance thresholds; the error-bar depicts the standard error. A point (x, y) on the figure should be read as "users issued, on average, y commands whose distance from the first command is larger than x over each 10-second period". Values $y \geq 1$ indicate that it is likely that users need an AoI switch *every* 10-second period. The results indicate that players often issue commands that switch the current screen (the current focus area), effectively switching their AoI.

This new phenomenon, of the presence of multiple, frequently changing AoIs per MAVE user, is an important motivation for the mechanism and system architecture we will introduce later. If we adopt the traditional AoI approach, which maintains only single AoI per player, choosing the size of AoI as the size of the screen will lead to significant AoI switching management overhead, and late delivery of states. Alternatively, the size of the single-AoI area could be very large, to cover all the possible areas of interest, thus leading to inefficient resource usage. We further show the inefficiency of the single-AoI approach via simulation in Section 6.2.

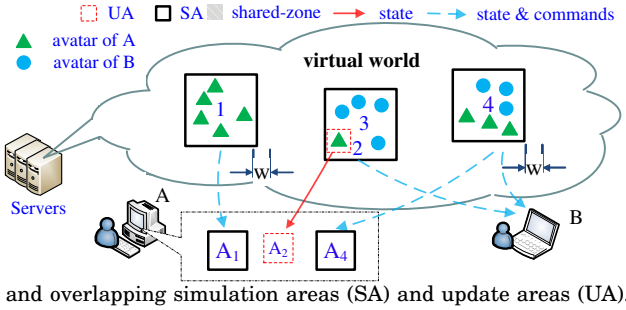

Fig. 3. A game map and overlapping simulation areas (SA) and update areas (UA). w is the width of the shared-zone.

4. AREA OF SIMULATION

In this section, we introduce a new scalability mechanism, the Area of Simulation (AoS). The key characteristic of our mechanism is the combined use of the EBLS and AoI, to efficiently maintain the areas of interest of each player.

The AoS mechanism adopts a distributed server architecture in which the virtual world (map) is divided into non-overlapping sub-maps. Each sub-map is simulated by one server and can also be simulated by clients. The AoS mechanism contains three parts: the partitioning of the virtual world into different types of areas, the mapping of areas, and the simulation of areas. We describe these three parts in the following, in turn.

4.1. Partitioning the Virtual World

From the user's point of view, the virtual world is partitioned into a number of areas. These areas can be areas of interest (AoI) or areas of non-interest (AoN). For each area of interest, depending on the operational model adopted, an area can be either a simulation area (SA) or an update area (UA).

For each SA, the game client receives events of that area, and then performs the simulation of that area (akin to EBLS-based operation). For virtual world objects and avatars in SAs, users have the most up-to-date and precise information. Currently, each sub-map can be operated as an SA. For each UA, the game client receives messages about the state-updates of that area, and updates the state of that area accordingly (akin to AoI-based operation). The visible area of a player's avatar can be a UA. For each player, a UA can overlap with the other UAs but not any SA. Users may receive different frequencies and precisions of the state-updates. Thus, for UAs, the user may have less up-to-date and lower-fidelity game-states than for SAs. For AoNs, the user will not receive any messages.

4.2. Mapping of In-game Areas to Real-World Resources

The *mapping* of an area, to either SA, UA, or AoN, depends on the user's interest, application logic, and resource availability. When the game client decides, based on measured or predicted interest, that the user should receive the most up-to-date information of an area, it classifies it as an SA. Similarly, areas of little or no interest are classified as UA and AoN, respectively. The lack of sufficient resources to manage SA or UA may force the game client to re-classify an area to a lower class.

An example of the AoS is illustrated in Figure 3. The virtual world contains sub-maps 1, 3, 4, and the other submaps. Player A has many avatars located in sub-maps 1 and 4, which are therefore classified as player A's SAs (A_1 and A_4). Player A also has an avatar exploring sub-map 3, the area visible to that avatar is a UA of user A (A_2). UA A_2 of user A is a special situation: although user A is highly interested in this area, user A already has two other SAs and, due to a lack of computational power, cannot afford to fully simulate another SA; instead, player A will use excess bandwidth

ALGORITHM 1: Operation of the AoS mechanism.

```
1: while not end-of-game do
2:     increment game tick;
3:     receive last turn's events issued in SAs;
4:     receive last turn's events issued in shared-zones surrounding SAs;
5:     receive last turn's states from shared-zones;
6:     receive summary of state updates of UAs;
7:     update game states according to DVEs logic by applying events to states of SAs;
8:     interpolate game states, display current states, and receive user's commands;
9:     advance simulation turn and send turn advance message, if all the events of a turn have
       been received;
10: end while
```

to get updates from the server responsible for area 2. We will further describe the management of areas in Section 5.2.

4.3. Simulation of In-game Areas

Because the virtual world is partitioned, events that can affect multiple areas raise a *data replication* problem for the AoS mechanism. Processing such a shared-event may require access to state information from all the areas the event affects. To provide users with the illusion of a seamless un-partitioned virtual world, the AoS adopts a *shared-zone* technique to manage data replication of areas. Areas that can be affected by shared-events maintain shared-zones that overlap with other areas. As Figure 3 shows, the shared-zones are the gray areas around each SA. We choose the width w of the shared-zone to be larger than the maximum effect range (MER) of the shared event. The MER can be pre-determined by application designers according to game logic. In all MAVEs we have surveyed, the MER is relative small comparing to the on-screen view (the maximal vision range of avatars in StarCraft is 14 tiles, while the screen width is 25 tiles). Each server will exchange with the others the states of shared-zones it manages for processing shared-events.

The state of the virtual world may be changed with the passage of time, and by different events and by distributed clients multiple times, even inside a simulation tick. Thus, knowing what the correct state is after the update has been made is the problem we are facing. The CAP theorem [Brewer 2012] states that Consistency, Availability, and Partition-tolerance cannot be simultaneously guaranteed in a distributed system. In AoS, different parts (partitions) of the virtual world are hosted in the Internet where *temporary partitions* of the network caused by latency or message loss are bounded to happen. During the period when the network is partitioned, we can either cancel the players' operations and thus decrease availability, or process the players' commands but with the risk of inconsistency. Similar to some previous work [Bernier 2001; Bharambe et al. 2008; McGee 2011] which treat consistency as a non-first requirement, we believe that the availability of DVEs (players receive responses for their operations in time) is more important than the consistency of DVEs. Thus, we do not ensure strong consistency as EBLS does. In AoS, we choose to partition the virtual world for scalability but at the cost of some inconsistency. In other words, we *trade off consistency for scalability* by allowing some game states to become inconsistent. By adopting this approach, we allow by design that the states of some parts of the partitioned virtual world may be different than the states that would have resulted from the same sequence of player commands executed in an un-partitioned virtual world.

The simulation operation of the AoS mechanism is described in Algorithm 1. For each simulation tick, each *client* receives the *commands* from the servers of SAs (line

Table I. Summary of mechanisms used in the AoS architecture.

Problem	Mechanism	Novelty
Change of users' interest	dynamic area management (Section 5.2)	new
Bandwidth consumption of UAs	forwarding pool and level of detail (Section 5.3)	adapted
Bandwidth consumption of shared-zones	delta encoding (Section 5.4)	re-used

3); additionally, the client receives all the commands issued in the shared-zones of the SAs (line 4). Each client also receives the *states* of the shared-zones (line 5), which enables processing the events that require information from the neighboring areas. For UAs, each client receives a summary of state-updates from the server managing each UA (line 6); depending on the resource availability of the server and on the game logic, the client may receive updates of various details and with various frequencies. After receiving all the needed information, the client will perform simulation to update its local view of the virtual world (line 7–8). During each tick, the *server* needs to read events from clients, read events and states of shared-zones from the other servers, sort the events, perform the simulation, and send the events and states to the clients.

For the simulation operation, as we trade off consistency for scalability, some inconsistency may happen compared to the un-partitioned simulation. We rely on the application developer to use additional techniques to mitigate this drawback [Chen and Verbrugge 2010; Bernier 2001]. For example, two avatars from different areas may collide unexpectedly. Application-specific logic, such as using distributed collision detection [Chen and Verbrugge 2010], allowing the bounding boxes used for collision detection to be slightly bigger than the actual size of the avatars, can be used. In this work, we simply disable collision detection following the practice of World of Warcraft (WoW) [McGee 2011]. In Section 6.3, we will show that the AoS mechanism does not introduce much inconsistency: most of the time, the inconsistency is unnoticeable.

5. AOS-BASED SYSTEM ARCHITECTURE

In this section, we integrate the AoS mechanism into an MAVE architecture. Table I summarizes the problems and the mechanisms (Section 5.2 to 5.4) adopted to solve them. Last we discuss the implications and limitations of our mechanisms in Section 5.5.

5.1. Architecture Overview

To operate the entire virtual world, our architecture comprises three types of logical nodes: the registration server, the clients, and the area servers. The registration server is responsible for the registration of the other types of logical nodes and to reply to queries to locate an area server. We describe the functions of the clients and of the area servers in the following.

Each client is responsible for managing the virtual world for a player. Clients can connect to multiple area servers. If a client has an SA which is managed by an area server, the client is a Simulation Area Client (SAC) for that area server. Similarly, if a client has a UA which is managed by an area server, the client is an Update Area Client (UAC) for that area server.

Each area server is responsible for managing an area and for communicating with clients. Each area server is an SAC for itself. It is responsible for receiving the commands issued by players in its area, for forwarding the commands in its area and the shared-zones' states to all its SACs, for forwarding its states to all its UACs, and for forwarding the selected state of its shared-zones and commands to neighboring area servers. An area server is neighboring to another area server only if the areas they manage are spatial neighbors in the virtual world.

5.2. Dynamic Area-Management Mechanism

Users may change their interest at run time. We propose a *dynamic area management* mechanism to adapt to the change of users' interest. The mechanism allows each user to have up to n SAs concurrently, where n can be predefined by the application developers according to application logic. To select the SAs and UAs for each player, our mechanism relies on a *dynamic and automatic ranking* of areas, using the *level-of-attention* (interest) shown recently by the player. The top-ranked n sub-map candidates are marked to become SAs, and the avatars' visible areas which are outside the selected SAs are marked to be UAs. In this work, the *level-of-attention* of a sub-map v is calculated as:

$$v = w_1 \times \sum_i (\frac{s_i}{s_t}) + w_2 \times \frac{t_a}{t} \quad 0 \leq s_i \leq s_t \quad 0 \leq t_a \leq t$$

which consists of two terms: the spatial value (left term) and the temporal value (right term). w_1 and w_2 is the relative weight for the spatial value and the temporal value, respectively ($0 \leq w_1, w_2 \leq 1$, $w_1 + w_2 = 1$). For the spatial value (left term), s_i is the score of the user owning an avatar with id i that is located in the sub-map, and s_t is the total score of all the avatars owned by the user. The intuition is that the more avatars gather in an area, the higher the player's interest in that area. The score s_i of avatar i can be assigned by MAVE designers according to MAVE logic. For example, in Age of Empires, a swordsman needs to be trained using a certain amount of in-game resources (e.g., 50 units of food, 20 units of gold), hence the score of a swordsman can be assigned as 70 (50+20). For the temporal value (right term), a player's interest in a sub-map is measured by the accumulated time t_a that the player has seen the sub-map rendered on-screen, at each time window t (i.e., 30). The intuition is that the more time a player has seen a sub-map, the higher the player's interest in that area. MAVE designers can tune the relative weights w_1 and w_2 according to their designs, for example, by setting w_2 higher, the MAVE can respond to players recent activities faster. In our system there are two types of players, human and artificial-intelligence (AI) players. For AI players, the temporal term is not taken into account ($t_a = t$). This mechanism is executed every t seconds (i.e., 30). To avoid the frequent changing of the level-of-attention ranking, a sub-map is an SA candidate, only if its level-of-attention is higher than a threshold th (i.e., 0.1).

There could be many ways to calculate the level-of-attention values by using spatial, temporal, social, and machine-performance metrics. For example, spatial metrics can include the location of the player's base, or the number of avatars present in the area, or the total amount of in-game resources invested by the user in that area. Temporal metrics can include the number of recently issued commands, or interaction history. Social metrics can include a summation of the interest shown by in-game allies. Machine-performance metrics can include a dynamic assessment of the computing and network capabilities of the player's machine. For example, a resource monitor can be integrated into each player's machine, once a machine cannot maintain the minimal simulation speed of a sub-map, the level-of-attention of the sub-map can be calculated as zero, which leads to the demotion of the sub-map to a UA. We leave further exploration of the calculation of level-of-attention as future work.

The dynamic ranking of areas allows for the creation and destruction of areas for each player. For example, in RTS games it is customary to build temporary bases with tens of mobile and immobile avatars; such temporary bases can lead to a temporary SA being created. As areas may be promoted to SA status, or demoted to UA or AoN status by the area management module, the MAVE operator can provision and allocate resources adapted to the player's needs, thus maintaining the quality of service.

For a client, upon losing interest in an SA, the area becomes first UA(s); further neglect leads to the conversion into an AoN, which makes the area server stop communicating with the client. Conversely, if the level-of-attention ranking of an area increases, a new SA needs to be created from an existing UA. First, the area server pauses the simulation procedure of the sub-map that contains the UA, and serializes the data for that sub-map. Then, the area server sends to the client all the needed state information and pending commands from the current simulation turn. Finally, the area server resumes the simulation and the gaming procedure is as usual. During this procedure, the area server needs to inform the neighboring area servers that the simulation of this sub-map is paused and the other neighbor area servers will not need to wait for the states sent by this sub-map. For interested readers, please refer to Appendix A for how the live-migration technique [Clark et al. 2005] is adapted into the AoS system to smoothen the UA-to-SA transitions.

When UA-to-SA or SA-to-UA transitions occur, depending on the current counts and limits concerning each area type, other areas may be demoted to or promoted from the status of SA and/or UA. To avoid possible cascading occurrence of transitions of UA-to-SA or SA-to-UA which may negatively impact the performance of the system, at each sub-map i, at most a_i (i.e., 50) UA-to-SA transitions and at most b_i (i.e., 50) SA-to-UA transitions are allowed by the server of the sub-map within t (i.e., 30) seconds. Obtaining a satisfactory setting of t, a_i, and b_i to avoid cascading occurrence of transitions may require experiment-tuning with a deployed DVE used by many users, which is out of the scope of this work.

5.3. UA State Dissemination Mechanisms

During the process of simulation, an area server needs to send state-updates to all connected clients, with high frequency. If an area is popular, the bandwidth consumption may exceed the capacity of the server managing this area. To alleviate this situation, we build a *forwarding pool (FP)* mechanism, in which the area server makes use of the idle upload bandwidth of the SACs. By making use of a unique property of the AoS mechanism, that all the SACs of an area have the same data as the area server, our FP can use some of the resource-rich SACs to help disseminating the states. In our current design, all the SACs (including the area server) of that area run a round-robin algorithm to select, in turn, an SAC to send state-updates to one UAC.

By using the FP mechanism, the upload-bandwidth consumption of servers can be greatly reduced. However, if there is no SAC or the aggregate upload bandwidth of the SACs cannot meet the demand of all the UACs, some form of state-reduction technique is needed, leading to less up-to-date states. We design a state-reduction mechanism, *level of detail (LoD)*, which effectively reduces network consumption by sending state-updates of different avatars at different frequencies, instead of a single fixed frequency. Each avatar i is assigned a score s_i which can be determined by the game designers according to the game logic (e.g., total amount of in-game resource used to train the avatar). Firstly, the avatars are sorted according to their scores in decreasing order, the state-updates of the top $p\%$ (e.g., 10) avatars will be sent every tick. Secondly, for the other $(100 - p)\%$ avatars, each avatar i has its own update frequency f_i (e.g., 0.025). Thirdly, when a user issues a command to an avatar whose id is k, the states of the avatars that surround avatar k will be sent to the user at every tick. After t_{LoD} (e.g., 150) ticks, if there are no further commands from the user which affect avatar k, the update frequencies of avatars around avatar k are restored to their normal update frequencies. p, f_i, and t_{LoD} can be assigned by application designers according to game logic. For example, the higher s_i is, the higher f_i would be. To obtain the optimal parameter setting of the LoD mechanism, we recommend that for complex games, a combination between game designer expertise and experimental tuning is needed. For

all the avatars whose state updates are not sent every simulation tick, our mechanism relies on techniques such as dead-reckoning [Bharambe et al. 2008; Bernier 2001] to interpolate/extrapolate the avatars' positions. The LoD mechanism promises to reduce the network consumption significantly, without reducing the accuracy of information about the avatars that the player is paying attention to.

5.4. Shared-zone State Dissemination Mechanism

The area servers of the AoS mechanism need to send states of their shared-zones to all their SACs. As we will show later in Section 6.2, the states can consume more than 20% of the server upload bandwidth. Thus, we use a *delta-encoding* technique [RFC3284 2002] to reduce the bandwidth consumption. Delta-encoding technique sends the difference of data instead of sending original data to client. When sending the shared-zones' state to clients, the area server will first get the difference of data, and then transfer only the difference of data to clients. If the states of shared-zones only changed slightly since the last state transfer, this technique can significantly reduce the bandwidth needed for sending states of shared-zones.

5.5. Implications and Limitations

The AoS mechanism gives DVE designers the ability to tune the system by configuring the trade-off between the high-fidelity, compute-intensive SAs and the relatively low-fidelity, network-intensive UAs. In this way, the AoS mechanism addresses for MAVEs the resource challenges in Section 5.5.1, and the consistency challenges in Section 5.5.2. We also discuss some limitations to the AoS mechanism in Section 5.5.3.

5.5.1. Resource Challenges. The AoS mechanism has good scalability, because only a few areas catch the interest of each player (see Section 3.2), so each client simulates only a few, high-interest areas,. In contrast to the EBLS mechanism, the AoS mechanism does not simulate the *entire* virtual world. Different from the traditional AoI mechanism, the AoS mechanism reduces the network consumption by transferring *both* commands and state-updates.

5.5.2. Consistency Challenges. Compared to update-based DVEs, such as WoW, which have server-side sequential consistency and client-side eventual consistency , the AoS mechanism provides sequential consistency for SAs, and eventual consistency for UAs and shared-zones, *both* on server- and client-side. Thus, the AoS mechanism can satisfy the requirement that the areas in which players show interest have high consistency guarantees.

5.5.3. Limitations. The AoS mechanism adds some complexity into the design of MAVE servers. A DVE designed traditionally for the single-server architecture, may need to be re-designed to adapt to the distributed-server architecture of the AoS mechanism. Moreover, the DVE designers may need to conduct experiments, to determine the area management parameter (n) needed to achieve optimal scalability for a specific DVE. However, as we will show in Section 6, even by setting $n = 1$, the AoS mechanism is more scalable than the other traditional models. Moreover, the distributed-server architecture is already the de facto standard in commercial game development.

The AoS mechanism guarantees only eventual (instead of sequential) consistency in shared-zones and UAs. This may dissatisfy users who frequently control avatars in such ares. Game designers can change the design of their DVEs, by reducing or limiting the chance that users are controlling avatars in the shared-zones (e.g., the border of each sub-map can be designed to be uninteresting to players). Moveover, game designers can adopt multiple levels of consistency-control protocols for different scenarios [Krammer et al. 2012; Zhang and Kemme 2011]. For example, strong consistency

Table II. Overview of experiments in Section 6.

Experiment	Evaluation target
Comparison with alternatives (Section 6.2.1)	Whether AoS scales under different scenarios
Comparison of Area Management mechanisms (Section 6.2.2)	Whether the dynamic management mechanism works
Forwarding pool and level-of-detail (Section 6.2.3)	The message reduction of these mechanisms
Consistency evaluation (Section 6.3)	Measuring the consistency tradeoff of AoS

Table III. Default experiment parameters.

Name	Meaning	Values
$w \times h$	virtual world size	1280×512 tiles
N	number of users	[10 to 400]
c	frequency of user's input	1 command per 10 ticks
K	number of avatars per user	50
n	maximum number of SAs per user	1
v	avatar speed	1 tile per tick
r	avatar vision range	10 tiles
w	width of shared-zone $w = r$	10 tiles
tps	number of ticks per second	40 ticks
nt	number of ticks per turn	2 ticks
sl	simulation length	10,000 ticks
w_1, w_2, s_i	level of attention parameters	$w_1 = w_2 = 0.5, s_i = 1$

protocols can be used for important scenarios, such as trading, which require strong consistency, whereas weak consistency protocols can be used for less important scenarios, such as avatar movement. We discuss next two possible solutions, the two-phase commit protocol and dead-reckoning.

The two-phase commit (2PC) protocol is a strong consistency protocol that can be used for important events of concurrent reading or writing on multiple servers. For the in-game trading example, assuming that an avatar needs to trade an item with another item located on another server, the 2PC protocol can be used to ensure the correctness of the trading operation. In our architecture, by using the 2PC protocol, the player's command would be delayed for an additional two round-trip times, and the number of messages required by the protocol would increase linearly with the number of nodes involved [Najaran et al. 2014]. Thus, the overhead of using the 2PC protocol would be acceptable for important events, which normally occur much less frequently than other events.

Dead-reckoning (DR) is a weak consistency protocol that can be used to hide, for avatar movement, the inconsistency due to the late-arrival of state-updates. DR interpolates and extrapolates the positions of an avatar, based on the avatar's location history and velocity. By interpolating, the movement path of an avatar between the two positions can be smoothed. By extrapolating, the position of an avatar can be predicted. Moreover, DR can be used to predict collisions [Chen and Verbrugge 2010]. The overhead of DR depends on the interpolation and extrapolation methods used [Yahyavi et al. 2012; Bharambe et al. 2008], such as linear kinetics based on Newton's second law, but is low in general. Thus, DR could be used as a weak consistency protocol in our architecture, with low overhead.

6. SIMULATION RESULTS

In this section, we evaluate AoS and four alternatives experimentally in a simulated environment. We present results obtained in a real-world environment in Section 7. Overall, our results indicate that AoS *is more scalable than the alternatives*.

We describe the experimental setup in Section 6.1. In Section 6.2, we compare AoS against four alternatives. The results show that AoS can achieve much lower network consumption than the pure update-based model (e.g., AoI), due to using the idle CPU resources on the client's side, and AoS can achieve much lower CPU consumption than the pure event-based model (e.g., EBLS), due to simulating only parts of a virtual world. In Section 6.3, we show that AoS achieves scalability without sacrificing too much consistency: 99.5% of the drift distance [Diot and Gautier 1999] of avatars can converge within about 0.3 seconds (a limit that is acceptable even for advanced users, see Section 2.1). Table II summarizes the experiments conducted in this section.

6.1. Experimental Setup

The default experiment parameters are shown in Table III. The simulation is running on a $1280 \times 512\,tile$ map, partitioned into 5×2 sub-maps of $256 \times 256\,tile$ each. The simulated virtual world map is 10 times larger than the largest game of StarCraft II.

This map size is consistent with our goal to scale this exemplary game and with the game-design scalability challenge (Section 3.1). As determining the maximum number of SAs that each user's machine can support is orthogonal to our work, we assume that each user can have up to n SAs.

Each player is assigned a base, uniformly, randomly distributed across the virtual world, with 50 avatars distributed around it. Each player will set the sub-map where the base locates in as an SA, and keep the area as an SA until the end of the simulation. The simulator is configured to update with a frequency of 40 ticks per second. Each avatar's vision is a square, centered on the avatar, with a range $r = 10$ tiles. The movement speed of the avatars is 1 tile per tick. The vision range and movement speed is similar to the setup of StarCraft II.

We run the simulation for 10,000 ticks (we have run some experiments with 50,000 ticks, and the results are similar). Unless otherwise specified, all the simulations are conducted with 60 users (about 4 times larger than the maximum number of players in one game of StarCraft II, which is 16) in a simulated LAN environment with no latency. Following the design of the very popular RTS game Age of Empires [Terrano and Bettner 2001], all the commands are scheduled to run 2 turns later.

Workload models: Modern MAVEs such as RTS games do not support more than 32 players in a game instance, so we are not able to obtain real-world workload traces with many users. Instead, based on our experience with popular RTS games [Terrano and Bettner 2001] and the code of a modern open-source RTS game engine [0 A.D. team 2014], we evaluate AoS against four different *workload models*. Each workload model is a combination of a *command model* and a *mobility model*. For the command model, each player is restricted to issue 1 command per 10 ticks, which is equivalent to 4 commands per second. This mimics player behavior during intense operations [Terrano and Bettner 2001]. Each command will order a randomly selected avatar to go to a position according to a mobility model. The *mobility models* are Weighted random Walk (WW), Weighted random walk Inside sub-map (WI), Weighted random walk with Distance (WD), and SAMOVAR. In WW, we partition the virtual world into multiple non-overlapping $16 \times 16\, tile$ areas, and randomly assign a weight w between 1 to 10 to each area. Each user is assigned i high-interest sub-maps, where i is sampled from the number of high-interest areas per user (with $h = 4$, see Figure 2 (left)). In WW, when a user commands an avatar to go to a new destination, the avatar selects a high-interest sub-map randomly. Then the avatar has a higher probability to go to a grid of the selected sub-map with higher weight, and it will go to a random position inside that grid. WI is similar to WW, but i is fixed to 1. In WD, the probability p to go to a grid is defined as $p = \frac{w}{d^2}$, where w is the weight (1 to 10) assigned to that grid, and d is the distance between the centroid of that grid and the player's base. For a player, this will make many avatars move in close proximity of the base, with only a few of the avatars going to some (valuable) spots away from the base. SAMOVAR is developed based on [Shen and Iosup 2014]. SAMOVAR acts similarly to WW, except that each user has a limited amount of grids to visit, and each user has different personal weights to visit those grids. Albeit we do not evaluate our system using real-users, the spatial AoI changes are higher in the 4 workloads than in StarCraft (see Appendix B.1), this suggests that the scalability results achieved by AoS will be better for the real-users than for the 4 workload models. Unless otherwise specified, WD is the default workload.

Metrics: We look at four metrics: the network bandwidth consumption (upload and download), the compute unit, and the drift distance which we define as follows. We count the number of messages sent/received as network bandwidth consumption. The compute unit is a reference value to estimate the CPU consumption. We calculate the compute unit, at each tick, as the number of avatars simulated at each client/server. We do not count the computation used for updating objects and for processing events

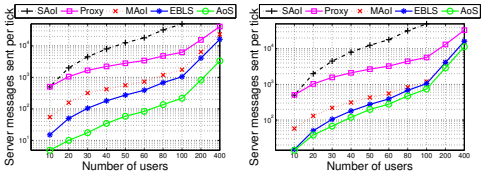

Fig. 4. Network upload: (left) the WI workload; (right) the WD workload. (Logarithmic scale on vertical axes)

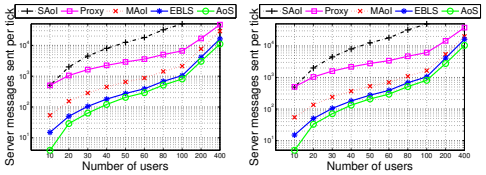

Fig. 5. Network upload: (left) the WW workload; (right) the SAMOVAR workload. (Logarithmic scale on vertical axes)

(according to our measurement in the prototype implementation, the time required to perform the simulation of an avatar is about 200 times higher than the time required to process state-updates/events). The drift distance [Diot and Gautier 1999] is used to evaluate the difference of avatar positions between the partitioned (AoS) and the unpartitioned (EBLS) models. We issue the same commands at the exact simulation time in AoS and EBLS and compare the distances of each avatar's position obtained through AoS and EBLS. For each experiment, we report the 99.5 percentile of drift distance. Each experiment is repeated 10 times, and the metrics shown are the average values.

6.2. Scalability Evaluation: Proposed Mechanisms

In this section, we evaluate AoS under a variety of scenarios. Our main findings are that (i) AoS consumes at least 30% less bandwidth than all the other alternatives for all the workload models; (ii) AoS requires an order of magnitude less computation than EBLS, for the server; (iii) the improved AoS (that is, AoS for which we enable the mechanisms introduced in Section 5) can further reduce bandwidth consumption by up to 60%.

6.2.1. Comparison with alternatives. We evaluate the computation and network consumption of AoS against four alternatives: *single-AoI (SAoI)*, *multiple-AoI (MAoI)*, *proxy-server (Proxy)*, and *EBLS* explained in the following. *SAoI*, *MAoI*, and *Proxy* are pure update-based models, while *EBLS* is a pure event-based model. For each pure update-based model, a distributed server architecture is adopted, for which each server is responsible for simulating a sub-map of the virtual world. *SAoI* adopts a single, static area-of-interest approach, whose area is the whole map. In *MAoI*, each player can have multiple area-of-interest, and each area is the visible area around the player's avatars. MAoI represents an extension of current AoI techniques, but, unlike our AoS, lacks the areas with event-based updates (the SAs). MAoI can be also viewed as AoS without any SAs. *Proxy* [Müller et al. 2005] acts similarly to *MAoI*, but each server needs to send the states that it simulates to the other servers, for synchronization (the original Proxy uses one AoI per player).

Figure 4 shows the upload bandwidth consumption of the server for SAoI, MAoI, Proxy, EBLS, and AoS under WI and WD, in turn for various user counts. For ease of reading the figure, we truncate the results of SAoI when the number of users is larger than 100. SAoI consumes significantly more bandwidth than the other models; this result complements the analysis in Section 3.2 that a single static AoI does not work well for MAVEs. Proxy consumes the second-most bandwidth compared to AoI, as Proxy needs to send additional messages to the other servers. It needs about 2 times and 1.5 times higher bandwidth than MAoI under WI and WD, respectively. AoS requires 30% up to 80% less upload bandwidth than EBLS, because the AoS servers only transfer messages that are relevant to the players instead of every message. Compared to MAoI, AoS consumes 40% up to 80% less bandwidth. This is because the AoS servers transfer commands besides state-updates to players, by making use of the players' idle

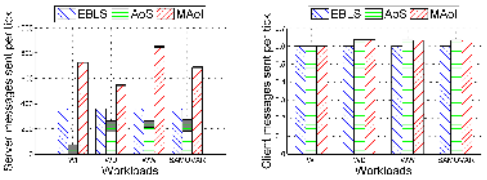

Fig. 6. Network upload, 60 users: (left) server; (right) client. (For the servers' upload of AoS and AoI, each bar is divided into 3 parts, from top to bottom (dark, gray, and light): number of commands sent, number of shared-zones' states sent, number of state-updates sent. Servers of EBLS only send commands, thus the bars for EBLS are not partitioned.)

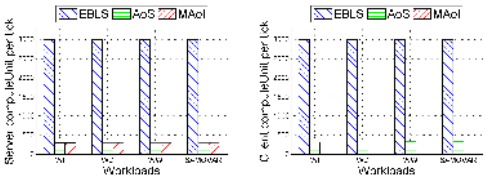

Fig. 7. Compute Unit under 4 workloads, 60 users: (left) server; (right) client.

CPU resources to perform simulation of the virtual world. Figure 5 shows the upload bandwidth consumption for WW and SAMOVAR. The results are similar to the results for WI and WD: SAoI and Proxy consume the most bandwidth, and AoS consumes the least bandwidth. The bandwidth consumption of AoS under WW and SAMOVAR is a bit more than that of under WI and WD, but AoS still consumes about 30% to 50% less than EBLS. As SAoI and Proxy consume much more bandwidth than the other models, we only consider EBLS, AoS, and MAoI in the rest of our experiments.

Dissecting upload messages: Figure 6 (left) shows the upload bandwidth consumption of the servers for EBLS, AoS, and MAoI from left to right, grouped by four workloads with 60 users. The servers of EBLS only send commands, while for AoS and MAoI, commands consume less than 5% of the network bandwidth, and the state-updates consume most of the bandwidth. For AoS, a large portion (more than 20%) of the bandwidth is used for sending the states of shared-zones. For MAoI, most ($\geq 95\%$) of its upload bandwidth is used for state-updates. Figure 6 (right) shows the upload of clients. The client upload is very low with less than 1 message per tick.

Computational overhead: Figure 7 shows the compute unit on the server-side (left) and client-side (right). On the server-side, as Figure 7 (left) shows, AoS and MAoI consume the same amount of compute unit, and EBLS consumes about 10 times more. The compute unit depends only on the number of avatars simulated. For AoS and MAoI, each server only needs to simulate a sub-map, on average, each server simulate 10% of the avatars, while for EBLS, the server needs to simulate all the avatars. Thus, the compute unit of AoS and MAoI are only 10% that of EBLS. On the client-size, as Figure 7 (right) shows, EBLS clients consume the same compute unit as the EBLS server. This is because EBLS needs to perform the same simulation of the whole virtual world both on the client-side and on the server-side. In AoS, each client is an SA client of a sub-map. The client performs the same simulation as the server of the sub-map. Thus, on average, AoS clients have the same amount of compute unit as that of AoS servers, that is, AoS clients consume about 10% of the computer unit of the EBLS clients. The computer unit of MAoI clients is 0, as MAoI clients do not perform any simulations.

6.2.2. Area management mechanisms and different numbers (n) of SAs. The previous experiments have shown that AoS with $n = 1$ is more scalable than the others. We show that by increasing n and adopting the dynamic area management mechanism proposed in Section 5.2, AoS can achieve much lower bandwidth consumption. We evaluate the impact of the area management mechanism, and the impact of the number of SAs per user by changing n from 1 to 4.

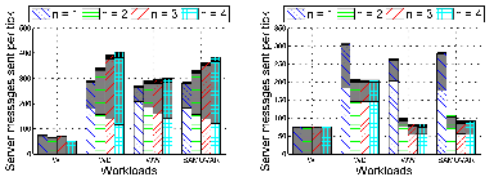

Fig. 8. Server messages sent (left) without and (right) with dynamic management. (each bar is divided into 3 parts, from top to bottom (dark, gray, and light): number of commands sent, number of shared-zones' states sent, number of state-updates sent.)

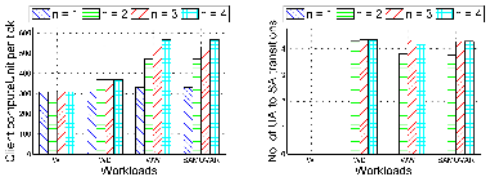

Fig. 9. Dynamic area management mechanism with increasing n: (left) client compute unit and (right) number of UA-to-SA transitions

Figure 8 (left) shows the results of using a static mechanism which randomly and statically sets SAs for each user. In contrast, Figure 8 (right) shows the results of using the dynamic area management mechanism (described in Section 5.2). In general, the number of state-updates sent for UAs (light bars in the figure) decreases with the increasing number (n) of SAs, while the number of commands sent (dark bars) increases.

As Figure 8 (left) shows, increasing n using the static mechanism slightly increases the bandwidth consumption for WD, WW, and SAMOVAR. This is because the number of shared-zone states that needs to be transferred increases with the increasing n, which decreases the bandwidth reduction through the use of more SAs.

As Figure 8 (right) shows, the dynamic mechanism achieves significant bandwidth reduction with increasing n. The amounts of shared-zones' states and state-updates sent by servers are both significantly reduced. For shared-zones' states, when a client has multiple neighboring SAs, the servers do not need to send the states of shared-zones of those neighboring SAs to the clients (the client itself has the master-copy of the states of shared-zones). The probability that a player will have multiple neighboring SAs is much higher when using the dynamic mechanism than when using the static mechanism, as the dynamic mechanism sets the top n sub-maps which contain more avatars of the player as SAs instead of randomly. For the state-updates, as most avatars are located in SAs for the dynamic mechanism, the servers send much fewer state-updates to clients than that of the static mechanism.

Increasing n using the dynamic mechanism does bring some overheads. The compute units used by servers do not increase, as each server only needs to simulate one sub-map regardless of n. For the clients, as Figure 9 (left) shows, the compute units increase with n. This is because with increasing n, each client has more SAs, which leads to increased simulation overheads. Figure 9 (right) shows the number of UA-to-SA transitions per sub-map. For WI, the number is zero, because the avatars of each player only move inside one sub-map, thus only one sub-map can be an SA for each player regardless of n. For the other workloads, on average, each player will experience about 4 UA-to-SA transitions per sub-map, that is, less than 1 transitions per sub-map per minute.

6.2.3. State dissemination mechanisms. To see whether the state disseminations we propose in Section 5.3 are efficient, we compare AoS (by default without any state dissemination mechanisms), with its variations that use state dissemination mechanisms: using forwarding pool (FP), using level of detail (LoD), and using both forwarding pool and level of detail (FP+LoD). When evaluating LoD and FP+LoD, for simplicity, we randomly pick $p = 10\%$ of the avatars whose state-updates are sent every tick, the others' state-updates are sent every 40 ticks ($f_i = 0.025$), and $t_{LoD} = 150$.

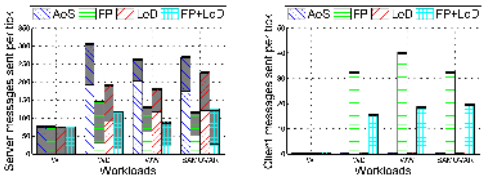

Fig. 10. Network consumption for AoS when using state dissemination mechanisms to distribute states: (left) server; (right) client.

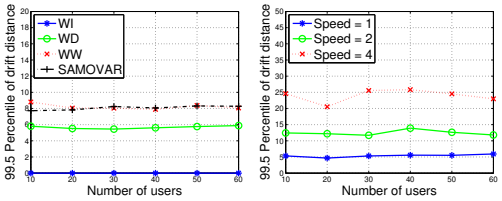

Fig. 11. Drift distance of AoS with: (left) different workloads and (right) WD workload and different movement speeds.

Figure 10 (left) shows that the server upload bandwidth consumption can be significantly reduced by using these state dissemination mechanisms. Using FP can lead to about 50% to 60% lower upload bandwidth consumption under WD, WW and SAMOVAR, because FP makes use of the idle clients' upload bandwidth to transfer the states of UAs. Using LoD can lead to about 15% to 30% lower bandwidth consumption for the server, but at the cost that some avatars' state-updates are less frequently transferred (less accurate). By using FP and LoD together, the server upload can be further reduced by about 20% to 50%, compared to using FP only.

FP increases the clients' upload bandwidth consumption, as Figure 10 (right) shows, the clients' upload bandwidth consumption for FP and FP+LoD increases from less than 1 message per tick to about 30 to 40 for WD, WW, and SAMOVAR. Because LoD transfers fewer states, the clients' upload bandwidth consumption for FP+LoD is lower than that for FP. We conduct experiments on LoD with varying p, f_i and t_{LoD}. We find that higher p, higher f_i, and higher t_{LoD} lead to higher bandwidth consumption.

6.3. Consistency Evaluation

In Sections 6.3.1 and 6.3.2, we evaluate the consistency tradeoffs of AoS using a reference metric: the drift distance of avatars (measured in tiles). In Section 6.3.3, we evaluate the percentage of collisions between sub-maps. Overall, we find that *AoS introduces negligible inconsistency and that resolving collision-generated inconsistency has low overhead.*

The source of drift distance is the distributed architecture. When an avatar is crossing the border of a sub-map, it needs to first send a command to the area-server of the sub-map where it will arrive. Then, the avatar will be moved to the future position, but in the neighboring sub-map. As a consequence, the avatar's position may be slightly different if the position is updated using AoS, when compared to EBLS. Because the median value of the drift distance across all the experiments in this section is *zero* (no drift distance experienced by 50% of the avatars), we analyze in the remainder of this section the presence of drift distance for extreme cases, by showing the 99.5 percentile of drift distance (*extreme drift distance*).

6.3.1. Workloads and speed. Figure 11 (left) shows the results for the 4 workloads. The extreme drift distance is small for these workloads. The extreme drift distance for WI with varying number of users is zero, because in this case avatars do not cross any border. The extreme drift distance for WW with varying number of users is about 8 tiles, but, because the movement speed of avatars is 1 tile per tick, this distance will eventually converge to zero within 8 ticks or approximately 0.25 seconds. Figure 11 (right) shows the impact of movement speed on the extreme drift distance: the higher the movement speed, the larger the extreme drift distance, but the extreme drift distance still converges to zero within 0.2 seconds.

6.3.2. Partition. We measure the extreme drift distance when partitioning the virtual world into finer sub-maps, from 64×64 to 256×256 tiles. The extreme drift distance is about 5 tiles for 256×256 sub-maps, but can increase to 20 tiles for 64×64 sub-maps. Partitions that increase the probability of avatars crossing the sub-map borders can lead to a significant increase of the extreme drift distance.

6.3.3. Collisions. We evaluate the overhead of imprecise collision detection, by measuring the percentage of collisions between sub-maps. We calculate this percentage by dividing the number of collisions between sub-maps by the total number of collisions. For each of the workloads, the percentage is low ($\leq 1\%$). The computational overhead needed to resolve imprecise collision detection depends on the method chosen to mitigate them, but in general it is low [Yahyavi and Kemme 2013]. For example, [Chen and Verbrugge 2010] use simple linear functions to predict and resolve collisions, a method whose computational overhead is low. Given the low chance of collisions between sub-maps and the low overhead of collision-detection protocols, we conclude that the overhead of resolving imprecise collision detection is low.

7. REAL-WORLD EXPERIMENTAL RESULTS

To demonstrate the applicability of AoS, we implement the architecture described in the previous sections and deploy the working system in a *real-world* environment. We evaluate the working system with a prototype RTS game, which represents the large-scale multi-player extension of an open-source, single-player RTS game [Granberg 2006]. This prototype game features many common elements of RTS game: training avatars, constructing buildings, fog-of-war, battles, etc.

We conduct real-world experiments an order of magnitude larger load than the current state-of-the-art, up to 100 users (instead of 16 users in StarCraft II) and 5,000 avatars involved in a large virtual world. The results show strong evidence that our AoS-based system is scalable.

7.1. Experimental Setup

Implementation: our system implementation[1] has about 25,000 lines of C++ code, which add to the about 7,000 lines of C++ code of the original RTS game. The AoS mechanism needs about 3,000 lines of code, while the other 15,000 lines of code are used to implement the multi-user part for the original single-player game. Our network module follows a similar design as 0 A.D [0 A.D. team 2014] and uses the reliable UDP library `ENet`[2]. We use the delta-encoding library `xdelta`[3] to encode the shared-zones' state to only transfer the difference of data, and `Zlib` for data compression. Without compression, the size of messages range from 40 bytes to 80 bytes. On average, the size of a command is 40 bytes, while the size of a state is 75. The size of messages lies within that of modern commercial RTS games [Claypool 2005].

Experimental environment: The experiments are conducted on the Amazon EC2 cloud. For our experiments, we use the "medium" instances of virtual machine (VM), each installed with Windows Server Datacenter edition 2008. Unless otherwise specified, we run 10 nodes (i.e., players) in each VM. All the other experimental configurations are the same as the default setup used in Section 6. Due to time and cost limitation, we use WI as the workload. As we focus on computing and networking resources, we disable the graphical output of the game.

Performance Metrics: We measure and report the ticks per second of each experiment, the bandwidth consumption of the payloads of network messages sent by the server and number of messages sent. As the number of messages sent by servers

[1]http://www.pds.ewi.tudelft.nl/~siqi/AoS.htm, [2]http://enet.bespin.org/, [3]http://xdelta.org/

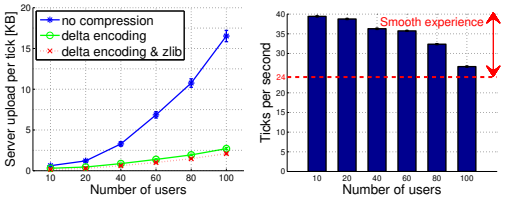

Fig. 12. Upload bandwidth (left) and tick per second (right).

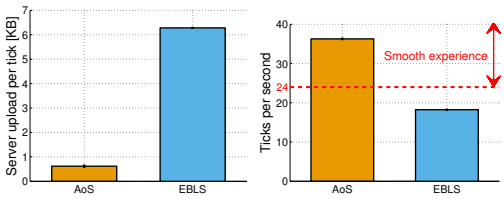

Fig. 13. Comparison between AoS and EBLS: upload (left) and tick per second (right).

matches well with the simulation results in Section 6, we only show the results for server upload bandwidth consumption and ticks per second (TPS). Each experiment is repeated 10 times, and the metrics shown are the average values.

7.2. Scalability Results

The bandwidth consumption of AoS scales linearly with the number of players. Figure 12 (left) depicts the upload bandwidth consumption of the server as a function of the number of users, for different methods of compressing the transferred data: AoS without compression, AoS with delta encoding, and AoS with delta encoding and zlib. Using the delta-encoding mechanism (proposed in Section 5.4) can reduce the bandwidth consumption significantly (about 80%), by only transferring the differences between the states of the shared-zones. Using zlib can further reduce the upload bandwidth consumption, by about 10%. Using both delta-encoding and zlib, AoS consumes about 2.1 KB per tick, when the number of players is 100. As the simulation speed is about 20-30 ticks per second, AoS consumes less than 75 KB per second.

Figure 12 (right) shows TPS as a function of the number of players. With an increasing number of players, the simulation overhead increases, which leads to lower TPS. When the number of players is 100, the TPS drops to 26. However, for all our experiments the TPS achieved in practice remains above the threshold required to deliver a smooth virtual experience (24 frames per second).

We compare AoS with EBLS using 40-node experiments. Because EBLS requires more memory and CPU resources than AoS, we run 5 (instead of 10) nodes per VM when running the prototype using EBLS. Figure 13 (left) shows the network consumption for these two mechanisms. The upload bandwidth consumed by the server is about 13 times lower for AoS than for EBLS; the message count for the server is similar, with the count for AoS about 15 times lower. This happens because, in the prototype, a state-update transfers more than an event. Figure 13 (right) shows the TPS results for the two mechanisms. Using AoS, the prototype can achieve a simulation speed of 36 TPS, whereas EBLS can only achieve 18 TPS. The 18 TPS is lower than the minimal TPS (24) required to achieve smooth virtual world experience, so the user experience will suffer when using an EBLS-based approach. *In summary, compared to EBLS, AoS can consume lower network bandwidth and achieve lower computation consumption, while still fulfilling all the requirements of MAVEs.*

8. RELATED WORK

Distributed Virtual Environments (DVEs) aim to generate a virtual world in which geographically distributed users can interact with others. There are two major operational models to update the DVEs in the clients' view: update-based and event-based. For the update-based model, the clients receive state-updates from the servers and then update their local view using the received information. For the event-based model, clients receive commands/events from a server and perform simulation, using the events to update the local game replicas. In our work, the DVE clients need to

receive state-updates and may perform simulation, which is different from cloud gaming techniques such as [Huang et al. 2013], where the DVE clients only receive video streams from a server. In this section, we compare AoS with the research on scalability in Section 8.1 and consistency in Section 8.2.

8.1. Scalability Techniques

Much recent research explores scalable DVEs. We identify four main approaches: zone-based, object-based, server replication, and interest management.

8.1.1. Zone-based. The virtual world is partitioned spatially into multiple non-overlapping zones; each zone is assigned to a separate server [Rosedale and Ondrejka 2003; Knutsson et al. 2004; Hu and Chen 2011]. Our *AoS*, SimMud [Knutsson et al. 2004], DSG [Lake et al. 2010], and MOPAR [Yu and Vuong 2005] partition the virtual world statically. In contrast, VSO [Hu and Chen 2011], Solipsis2 [Frey et al. 2008], Cell [Deng and Lau 2014], and [Lui and Chan 2002] partition the virtual world dynamically. As an example of zone-based DVEs, SimMud [Knutsson et al. 2004] partitions the game world into static zones and uses a peer-to-peer multicast channel to send game updates to clients. Many previous studies focus on scalable messaging, and do not consider game-logic processing. For the studies that do, the operational model is mostly update-based.

8.1.2. Object-based. The virtual world objects are load-balanced across servers [Morillo et al. 2007; Lu et al. 2006; Waldo 2008]. Each server is responsible for the simulation of a subset of objects (often called active objects), while the remaining ones (often called shadow objects), which are active in the other participating servers, are synchronized across servers. Proxy server [Müller et al. 2005] used for comparison in Section 6.2.1 belongs to this category.

8.1.3. Server replication. The virtual world states are fully-replicated at each server; clients connect usually to the closest server. The event-based model is usually adopted to reduce the network bandwidth. In this model, the events are transferred to some servers or broadcasted to all the servers, and the server performs the simulation based on events. EBLS is one of the most widely used server replication techniques adopted by DVEs such as [Zhang and Tang 2011][Cronin et al. 2004], and RTS games. As the states are fully-replicated, maintaining the state for a large virtual world is problematic for most servers.

8.1.4. Interest management (IM). IM determines information that is interesting and should be received by players [Yahyavi and Kemme 2013]. IM can be class-based or space-based. For class-based IM, users only receive specific types of information that are predefined per class, while space-based IM is based on proximity. AoI is a form of space-based IM in which a player only receives the information close to the location of the avatar(s) of the player. Usually, when an avatar moves, the AoI moves with the avatar. An AoI can be a zone [Knutsson et al. 2004], a geometry area inside a zone [Boulanger et al. 2006], or intersects with multiple zones [Yu and Vuong 2005]. The shape of an AoI can be: tile-based [Boulanger et al. 2006], circular [Ahmed and Shirmohammadi 2009], and free format [Bharambe et al. 2008]. The size of an AoI can be static [Hu and Chen 2011] or dynamic [Keller and Simon 2003]. Donnybrook [Bharambe et al. 2008] proposes an estimation of FPS players' interest based on distance between avatars, the aiming of the player's weapon, and interaction history. They classify avatars (based on interest) into two sets. Up-to-date states of the avatars in one set are received every frame while the states of the others are received every second. Different from previous work, we consider each player having multiple AoIs instead of one, and each AoI is operated using event-based or update-based model.

AoS is the first approach that combines zone-base partition, event-based and update-based models to support large-scale MAVEs, especially for RTS games.

8.2. Consistency Control

Pessimistic and *Optimistic* methods are two major classes of consistency control methods. *Pessimistic methods* anticipate inconsistency between data replicas when performing local actions. In contrast, *optimistic methods* assume no inconsistency exists and perform local actions instantaneously.

8.2.1. Pessimistic methods. The local-lag [Mauve et al. 2004] mechanism delays the execution of operations to reduce the probability of the occurrence of inconsistency. It usually delays commands according to a system-level delay value. The local-lag mechanism works efficiently when the value is larger than the largest network latency. Distributed transaction techniques such as two-phase commit (2PC) protocol are used in DVE. For example, [Najaran et al. 2014] adopt the 2PC method to manage distributed data by synchronizing events that span multiple servers.

8.2.2. Optimistic methods. AoS can be classified as an optimistic method which allows that inconsistency exists in shared-zones. There are two major classes of optimistic methods: time warp (TW) and dead reckoning (DR). In TW, all replicas are allowed to execute update optimistically, and the method needs to record old states and roll back when inconsistency happens. Although there are improved versions of TW such as trailing state synchronization [Cronin et al. 2004] and [Ferretti 2008], TW requires roll-backs, which may dissatisfy players; thus, we do not adopt this approach. DR, widely applied in DVEs such as [Bharambe et al. 2008], is a technique to reduce the network consumption for position updates. DR can be used in the AoS architecture to reduce the network consumption of UAs.

Other approaches exist. [Lupei et al. 2010] use the software transaction memory technique (STM) to build an FPS game which runs in a multi-core machine. STM may have high overheads and transaction abort rates; thus, fine tuning of transactions and even redesign of game logic are needed. [Zhang and Kemme 2011] and [Krammer et al. 2012] use different consistency protocols for different events. [Tang and Zhou 2010] use multiple update frequencies for avatars to improve time-space consistency. Their work can be used in ours to resolve the inconsistencies and reduce network consumption.

9. CONCLUSION AND FUTURE WORK

Multi-Avatar Virtual Environments (MAVEs) such as RTS games entertain millions of people in small-scale, non-communicating game instances of only 8–16 players. To enable a future generation of MAVEs, in this paper we investigate a new mechanism and a system architecture built around it, which are scalable and have many desirable properties.

Our main contribution is five-fold. Firstly, we conduct the first empirical investigation into the presence of areas of interest in MAVEs. We find that, unlike the other virtual environments such as RPG and FPS games, in MAVEs users have multiple areas of high interest and that interest location changes quickly. Secondly, our AoS mechanism is novel in its use of update-based and event-based operation for areas of interest and provides a versatile scalability-consistency trade-off. For the latter, the AoS mechanism ensures that only areas of high interest are fully simulated, that areas of limited interest only receive infrequent updates, and that areas of no interest do not consume either computational or network resources. Thirdly, we propose an AoS-based system architecture for scalable MAVEs, which supports the dynamic management of multiple areas of interest and several more common, scalability-related techniques. Fourthly, we implement this architecture as a real-world system, which is able to run

RTS games up to 100 users and 5,000 avatars in the same virtual world. Fifthly, we compare qualitatively and quantitatively our approach with various state-of-the-art approaches, and show strong evidence that AoS-based approaches offer superior performance and more flexibility for MAVEs.

For the future, we plan to investigate automatic tuning and balancing AoS-based systems, and integrate the AoS architecture with cloud computing techniques.

References

0 A.D. team. 2014. A free, open-source game of ancient warfare. http://wildfiregames.com/0ad/. (2014).

D. Ahmed and S. Shirmohammadi. 2009. Zoning Issues and Area of Interest Management in Massively Multiplayer Online Games. In *Handbook of Multimedia for Digital Entertainment and Arts*.

N. E. Baughman and B. N. Levine. 2001. Cheat-proof Playout for Centralized and Distributed Online Games. In *IEEE Conference on Computer Communications*. 104–113.

Y. W. Bernier. 2001. Latency compensating methods in client/server in-game protocol design and optimization. In *Game Developers Conference*.

A. R. Bharambe, J. R. Douceur, J. R. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang. 2008. Donnybrook: enabling large-scale, high-speed, peer-to-peer games. In *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. 389–400.

J.-S. Boulanger, J. Kienzle, and C. Verbrugge. 2006. Comparing interest management algorithms for massively multiplayer games. In *Workshop on Network and Systems Support for Games*. 1–6.

E. Brewer. 2012. CAP Twelve Years Later: How the "Rules" Have Changed. *Computer* 45, 2 (Feb. 2012), 23–29.

M. Buro and D. Churchill. 2012. Real-Time Strategy Game Competitions. *AI Magazine* 33, 3 (Sept. 2012), 106–108.

T. Chen and C. Verbrugge. 2010. A protocol for distributed collision detection. In *Annual Workshop on Network and Systems Support for Games*. 1–6.

C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. 2005. Live Migration of Virtual Machines. In *Symposium on Networked Systems Design & Implementation*. 273–286.

M. Claypool. 2005. The effect of latency on user performance in Real-Time Strategy games. *Computer Networks* 49, 1 (Sept. 2005), 52–70.

E. Cronin, A. R. Kurc, B. Filstrup, and S. Jamin. 2004. An Efficient Synchronization Mechanism for Mirrored Game Architectures. *Multimedia Tools Appl.* 23, 1 (May 2004), 7–30.

Y. Deng and R. W. H. Lau. 2014. Dynamic Load Balancing in Distributed Virtual Environments Using Heat Diffusion. *ACM Trans. Multimedia Comput. Commun. Appl.* 10, 2 (Feb. 2014), 16:1–16:19.

C. Diot and L. Gautier. 1999. A distributed architecture for multiplayer interactive applications on the Internet. *IEEE Network* 13, 4 (Aug. 1999), 6–15.

ESA. 2012. Essential Facts About the Computer and Video Game Industry: Sales, Demographics, and Usage Data. (Jul 2012).

S. Ferretti. 2008. A synchronization protocol for supporting peer-to-peer multiplayer online games in overlay networks. In *International conference on Distributed event-based systems*. 83–94.

G. Fiedler. 2010. What every programmer needs to know about game networking. (2010). http://bit.ly/7jSZl5.

D. Frey, J. Royan, R. Piegay, A. Kermarrec, F. Le Fessant, and E. Anceaume. 2008. Solipsis: A decentralized architecture for virtual environments. In *Workshop on Massively Multiuser Virtual Environments*. 29–33.

J. S. Gilmore and H. A. Engelbrecht. 2012. A Survey of State Persistency in Peer-to-Peer Massively Multiplayer Online Games. *IEEE Trans. Parallel Distrib. Syst.* 23, 5 (April 2012), 818–834.

C. Granberg. 2006. *Programming an RTS Game With Direct3d*. Charles River Media, Hingham, MA.

J. Gregory. 2009. *Game Engine Architecture*. A K Peters, Ltd., Natick, MA.

S.-Y. Hu and K.-T. Chen. 2011. VSO: Self-organizing Spatial Publish Subscribe. In *Self-Adaptive and Self-Organizing Systems*. 21–30.

C.-Y. Huang, C.-H. Hsu, Y.-C. Chang, and K.-T. Chen. 2013. GamingAnywhere: an open cloud gaming system. In *ACM Multimedia Systems Conference*. 36–47.

J. Keller and G. Simon. 2003. Solipsis: A Massively Multi-Participant Virtual World. In *International Conference on Parallel and Distributed Processing Techniques and Applications*. 262–268.

B. Knutsson, H. Lu, W. Xu, and B. Hopkins. 2004. Peer-to-peer support for massively multiplayer games. In *IEEE Conference on Computer Communications*. 96–107.

L. Krammer, G. Schiele, D. Koch, and C. Becker. 2012. Quality of Experience-Aware Event Synchronization for Distributed Virtual Worlds. In *IEEE International Conference on Parallel and Distributed Systems*. 604–611.

D. Lake, M. Bowman, and H. Liu. 2010. Distributed scene graph to enable thousands of interacting users in a virtual environment. In *Workshop on Network and Systems Support for Games*. 1–6.

H. Liu, M. Bowman, and F. Chang. 2012. Survey of state melding in virtual worlds. *ACM Comput. Surv.* 44, 4 (Aug. 2012), 21:1–21:25.

F. Lu, S. Parkin, and G. Morgan. 2006. Load Balancing for Massively Multiplayer Online Games. In *Workshop on Network and Systems Support for Games*. 1–6.

J. C. S. Lui and M. F. Chan. 2002. An Efficient Partitioning Algorithm for Distributed Virtual Environment Systems. *IEEE Trans. Parallel Distrib. Syst.* 13, 3 (March 2002), 193–211.

D. Lupei, B. Simion, D. Pinto, M. Misler, M. Burcea, W. Krick, and C. Amza. 2010. Transactional memory support for scalable and transparent parallelization of multiplayer games. In *European Conference on Computer Systems*. 41–54.

M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. 2004. Local-lag and timewarp: providing consistency for replicated continuous applications. *IEEE Trans. Multimedia* 6, 1 (Feb. 2004), 47 – 57.

J. McGee. 2011. The pros and cons of collision detection. http://wow.joystiq.com/2011/07/10/breakfast-topic-the-pros-and-cons-of-collision-detection/. (2011).

P. Miller. 2011. Professional Gamers: A Day in the Life. PCWorld online article. http://www.pcworld.com/article/221388/professional_gamers_a_day_in_the_life.html. (2011).

P. Morillo, S. Rueda, J. M. Orduña, and J. Duato. 2007. A Latency-Aware Partitioning Method for Distributed Virtual Environment Systems. *IEEE Trans. Parallel Distrib. Syst.* 18, 9 (Sept. 2007), 1215–1226.

J. Müller, J. H. Metzen, A. Ploss, M. Schellmann, and S. Gorlatch. 2005. Rokkatan: scaling an RTS game design to the massively multiplayer realm. In *International Conference on Advances in computer entertainment technology*. 125–132.

M. T. Najaran, S.-Y. Hu, and N. C. Hutchinson. 2014. SPEX: Scalable Spatial Publish/Subscribe for Distributed Virtual Worlds Without Borders. In *ACM Multimedia Systems Conference*. 127–138.

RFC3284. 2002. RFC3284: The VCDIFF Generic Differencing and Compression Data Format. (2002). http://tools.ietf.org/html/rfc3284.

P. Rosedale and C. Ondrejka. 2003. Enabling Player-Created Online Worlds with Grid Computing and Streaming. Gamasutra Resource Guide. (2003).

S. Shen and A. Iosup. 2014. Modeling Avatar Mobility of Networked Virtual Environments. In *Workshop on Massively Multiuser Virtual Environments*. 1–6.

S. Shen, O. Visser, and A. Iosup. 2011. RTSenv: An experimental environment for real-time strategy games. In *Workshop on Network and Systems Support for Games*. 1–6.

X. Tang and S. Zhou. 2010. Update Scheduling for Improving Consistency in Distributed Virtual Environments. *IEEE Trans. Parallel Distrib. Syst.* 21, 6 (June 2010), 765–777.

M. Terrano and P. Bettner. 2001. 1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond. (2001). Game Developer Conference.

J. Waldo. 2008. Scaling in games & virtual worlds. *ACM Queue* 51, 8 (Aug. 2008), 38–44.

A. Yahyavi, K. Huguenin, and B. Kemme. 2012. Interest modeling in games: the case of dead reckoning. *Multimedia Systems* 16, 3 (June 2012), 255–270.

A. Yahyavi and B. Kemme. 2013. Peer-to-Peer Architectures for Massively Multiplayer Online Games: A Survey. *ACM Comput. Surv.* 44, 4 (Sept. 2013), 21:1–21:25.

A. Yu and S. T. Vuong. 2005. MOPAR: a mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games. In *ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*. 99–104.

K. Zhang and B. Kemme. 2011. Transaction Models for Massively Multiplayer Online Games. In *IEEE Symposium on Reliable Distributed Systems*. 31–40.

L. Zhang and X. Tang. 2011. The Client Assignment Problem for Continuous Distributed Interactive Applications. In *International Conference on Distributed Computing Systems*. 203–214.

Online Appendix to: Area of Simulation: Mechanism and Architecture for Multi-Avatar Virtual Environments

SIQI SHEN, ALEXANDRU IOSUP, DICK EPEMA, Delft University of Technology, The Netherlands
SHUN-YUN HU, Academia Sinica, Taiwan, R.O.C.

In this appendix, we discuss in Section A how to integrate a live-migration technique to make the UA-to-SA transitions smooth, present in Section B more experimental results, and discuss in Section C how to alleviate the impact of Internet latency.

A. UA-TO-SA TRANSITION

We adapt the live migration technique proposed in [Clark et al. 2005], to make the UA-to-SA transitions smooth from the perspective of the player. The procedure, which we depict in Figure 14, consists of the following rounds:

(1) The SA server takes a snapshot of the sub-map it manages. The snapshot is transferred to the client. (round 1)
(2) While the snapshot is transferred to the client, the state of the sub-map may change. If this happens, the difference between the previous snapshot and the current game state is recorded, then, transferred to the client. (round 2)
(3) While the snapshot differences are transferred to the client, the sub-map may be changed again; the new difference will be recorded, but re-transferred only when needed. We limit the number of re-transfers to at most n (in our experiments, $n = 3$ delivers good results with low overhead) and only if the size of the information to be exchanged exceeds a threshold t (i.e., 20KB). (round 3).
(4) The server stops the simulation, transfers the final difference of the sub-map to the client, waits for the client's acknowledgement, and, last, resumes the simulation. The client becomes an SA client from this moment on (round 4).

As is depicted in Figure 14, the total time used for a UA-to-SA transition includes the time needed to transfer the snapshot, plus the times needed to transfer the differences of the snapshot. In round 4, the final round, the server will be stalled (paused). The stall time is the time to transfer the final difference of the sub-map, plus one round-trip time between the server and the client. Round 2 and each repetition of Round 3 are used to transfer the differences during the time of their either last round or last repetition. In Round 2 and in each repetition of Round 3, the size of the differences will be smaller than its last round or last repetition. Through these rounds, the size of the final difference of the sub-map becomes smaller, and thus, the stall time is shorter[2].

To evaluate the performance of the UA-to-SA transition, we conduct experiments in simulation and with the prototype. For simulation, the default snapshot size is configured as 1 MB, the default snapshot changing rate is 1 KB/s, and the one-way latency between the server and the client is 50 ms. These simulation parameters are realistic, within the range of modern RTS games. For comparison, we setup an Age of Empires HD (AoE) game (released in 2013), with 6 players each with 50 avatars[3]. The first

[2]It is possible that the sub-map dramatically changes during the transition, which increases the size of the differences for the next data transmission, but this is rare (or even impossible for typical DVE designs).
[3]in Section 6 and 7, on average, each sub-map has 6 player with 50 avatars each, by default.

DOI: http://dx.doi.org/10.1145/0000000.0000000

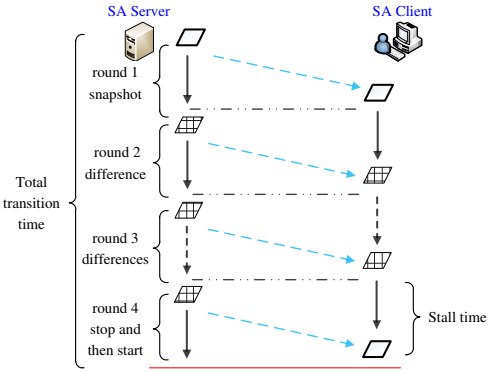

Fig. 14. Live UA-to-SA transition.

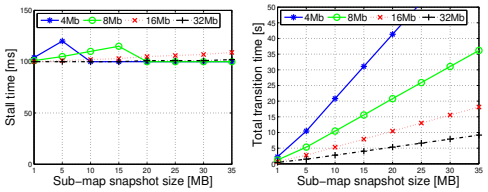

Fig. 15. UA to SA transitions with different sub-map sizes: (left) stall time and (right) total transition time.

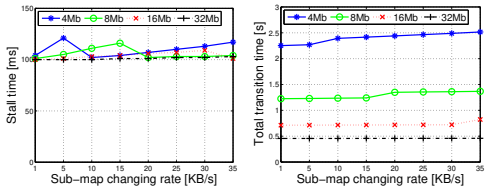

Fig. 16. UA to SA transitions with different sub-map changing rates: (left) stall time and (right) total transition time.

snapshot of the game is 0.62 MB. The snapshot of the game slightly increases during the game, and one hour later, the size of the snapshot reaches 0.98 MB. Assuming that the snapshot increase linearly over time, the snapshot changing rate of AoE is less than 1 KB per second. For our prototype with the same setup, the snapshot size is lower than 100 KB and its changing rate is about 1 KB per second.

The simulation results show that the stall time of the live UA-to-SA transition is very low, less than 140 ms. That is, a command given by a player to an avatar located within the sub-map may under the most unfavorable conditions be delayed for an additional 140 ms, which is acceptable latency for RTS games [Claypool 2005]. Figure 15 shows the stall time and the total transition time as functions of the snapshot size, for different client download bandwidth, ranging from 4 Mbit/s to 32 Mbit/s. Figure 15 (left) shows that the stall time is very small ($\leq$ 140 ms) for all the realistic sub-map snapshot sizes we have tried. Figure 15 (right) shows the total transition time, which increases with the increasing snapshot size. Figure 16 shows the stall time and the total transition time as functions of the sub-map changing rate. As indicated by Figure 16 (left), the stall time remains small ($\leq$ 140 ms) for all the realistic changing rates we have tried. The total transition time, as is displayed in Figure 16 (right), increases only slightly, when the sub-map changing rate increases; overall, the transition time remains low ($\leq$ 2.5 s) for all changing rates we have tried. For experiments with the prototype, the server and the client are located on the same computer, the bandwidth between them is capped at 16 Mbit/s, and the latency between the server and the client is artificially set to 50 ms. The results show that the stall time is under 110 ms and that the total transition time is under 215 ms. We also conduct experiments with varying n and t, and find that they do not have a significant impact on the performance of the UA-to-SA transitions.

B. SCALABILITY EVALUATION

We evaluate in Section B.1 how frequently players change their AoIs, and in Section B.2 the impact of 2 workloads on network performance.

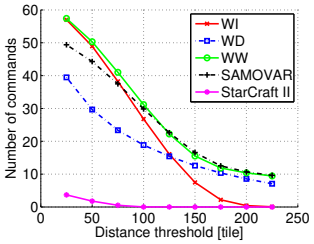

Fig. 17. Dynamics of interest.

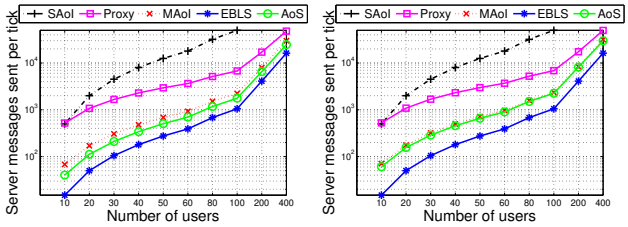

Fig. 18. Network upload: (left) the WW 50% workload; (right) the WWG workload. (Logarithmic scale on vertical axes)

B.1. AoI changes

The performance of AoS can be affected by how frequently the players change their AoIs. In this section, we study how frequently the AoI changes for the 4 workloads used throughout this work: WI, WD, WW, and SAMOVAR. We apply the same analysis as we have used in Section 3.2, Figure 2 (right). The analysis divides each player's command history into 10-second-windows, and identifies in each window the first command (if it exists). It then counts for each 10-second-window, the number of commands whose distance to the first command is longer than certain threshold (e.g., 25 tiles, which means about one full screen). The higher the number of commands far from the first command, the more the AoI changes spatially. The results for the 4 workloads and for the StarCraft trace, are depicted in Figure 17. On average, WI, WD, WW, and SAMOVAR have 40 to 55 commands over the distance threshold of 25 tiles, whereas the value for StarCraft is less than 4. This indicates that the AoI changes of the 4 workloads are much higher than that of the StarCraft trace, which gives a strong indication that, with real-human players, the scalability results of AoS will be even better than the results suggested by experiments with the 4 workloads.

B.2. Workloads

We further investigate the impact of the workload on network performance. To this end, we consider two variations of WW: WW-50 and WWG. These two workloads differ in the preference of avatars of visiting high-interest areas. For WW-50, about 50% of the destinations of the avatars will be located inside the high-interest sub-maps of each player. For WWG, each avatar will go to any grid of the map according to the grid's weight. WWG is an extreme case, in which the players do not have *any* high-interest areas. Figure 18 (left) shows the results for WW-50, for WW-50, AoS consumes less network bandwidth than the others, except EBLS. AoS consumes about 20% to 40% less bandwidth than MAoI. Compared to WW (shown in Figure 5 (left)), AoS consumes more network bandwidth under WW-50. This is because, with more avatars moving outside the SAs of players, more state-updates need to be sent. Figure 18 (right) shows the results for WWG. Similarly to WW-50, AoS consumes the least bandwidth, except EBLS. AoS consumes about 5% to 20% less bandwidth than MAoI. Under WWG, the benefit of message reduction brought by AoS is lower than in other cases. This happens because the avatars do not have any preference to visit the high-interest sub-maps, thus, AoS does not significantly reduce the network consumption.

C. INTERNET LATENCY

In this section, we propose a mechanism, Dynamic Delay-Execution (DDE), to alleviate the impact of Internet latency, and we show that the DDE mechanism can keep both the response time and the stall time low, and thus can fulfill the technical requirements for good user experience.

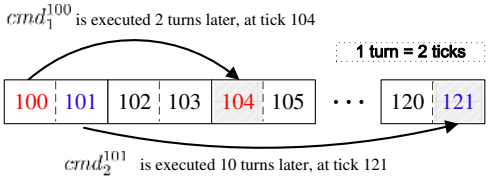

Fig. 19. Dynamic delay-execution.

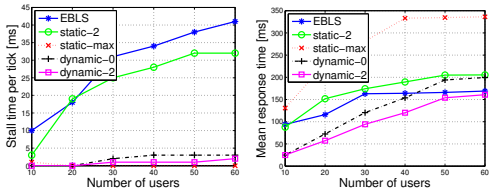

Fig. 20. Performance of EBLS and AoS with different delay-execution mechanisms: (left) Stall time and (right) Response time.

C.1. Dynamic Delay-Execution Mechanism

To be certain of having obtained all the needed information, both the server and the clients need to wait until the end of the current turn. Because messages are sent through the Internet, the update of the virtual world is slowed down by the slowest Internet connection. To alleviate the delays introduced by the Internet, we propose the DDE mechanism, which overlaps the communication with the simulation updates, by delaying commands issued during the current simulation turn to be executed not in the next turn, but after x more turns.

The *DDE* mechanism allows *each* client to have a *different* execution delay (ED) for each of the area servers. The *DDE* is an improvement of the traditional delay-execution mechanism [Terrano and Bettner 2001], which sets a single system-level delay for all clients. For the traditional delay-execution mechanism, larger delays ensure better tolerance to Internet latency, but make the game less responsive and slow down the advancement of virtual clock. The *DDE* can make the advancement of the virtual clock without any slowdown, and ensures the responsiveness of users' commands. The schematic plot of the dynamic-delay execution mechanism is depicted in Figure 19. As shown, cmd_1^{100} issued at tick 100 by user 1 will be executed at tick 104, and cmd_2^{101} issued by user 2 at tick 101 will be executed 20 ticks later after it is issued.

We describe how to calculate the ED. The commands issued from each client need to arrive at the server. Then the server sorts all received commands and sends the sorted commands back to clients. The time period between the moment when the commands are issued and when they are executed is at least the round-trip time. Thus, the ED x_{c_i,s_j} between client i and server j is calculated by $x_{c_i,s_j} = \lceil l_{c_i,s_j} \times 2 \div T \rceil$, where l_{c_i,s_j} is the one-way latency between client i and server j, T is the simulation turn duration, and $\lceil\rceil$ is the ceiling function.

As commands issued during the same virtual tick may be scheduled to be run in different ticks, the commands from clients with low latency may be executed faster. For fairness, the area server imposes a fairness constraint F, to limit the advantage gained by faster Internet connection, and defined as the maximal gap between the smallest and the largest ED of all the clients of an area server. Thus the execution orders of commands issued by all clients of an area will diverge by at most F turns. For example, by setting $F = 5$, cmd_1^{100} in Figure 19 will be executed 5 turns later instead of 2, while cmd_2^{101} remains the same. By default, we set $F = 0$. Any event/state-update that is issued in one area server and needs also to be processed by a neighbouring area server, will also be delayed. The ED of events/state-updates exchanged between server s_1 and s_2, x_{s_1,s_2}, can be calculated as $\lceil (a + b) \div T \rceil$, where a is the largest clients to servers latency (one-way), b is the inter-server latency between s_1 and s_2, and T is the simulation turn duration.

The one-way latency between two nodes is estimated using the mean value of multiple round-trip times divided by two. It may happen that some nodes experience temporary high jitter. For nodes experiencing high jitter (e.g., the jitter is higher than 0.2 of mean round-trip time), the latency is calculated as the latest largest measured value

instead of the mean value. The DDE mechanism runs periodically (e.g., every 30 seconds) to reflect the change of Internet latency. Each server needs to measure its latency between clients and determine the ED for each client.

Using the DDE mechanism, the players with lower latency still have unfair advantages to execute their operations. However, with the DDE mechanism, users from different parts of the world can join the DVEs and play with some fairness guarantee, instead of simply kicking a player if the players connection is laggy (as in the game OpenTTD), or no guarantee (as in WoW).

C.2. Experiments using Internet latency traces

We evaluate the impact of latency on stall time and response time, for AoS with different delay-execution mechanisms, and for EBLS, under the WD workload. We show that *the DDE mechanism can achieve negligible stall time and keep the response time low. Thus, AoS can work well in an Internet-based environment.* As MAoI used for comparison has the same command processing module as AoS in the server-side, all the metrics shown in this section are the same for MAoI and for AoS.

We compare AoS with the DDE mechanism to AoS with two static delay-execution mechanisms: static-2 and static-max. Static-2 delays all the commands for 2 turns (the default AoS setting in LAN), while static-max mechanism delays all the commands by a duration which is the largest round-trip latency divided by the turn duration. For the DDE mechanism, we use two configurations by setting the fairness parameter $F = 0$: dynamic-0 and $F = 2$: dynamic-2.

The simulation environment is the same as Section 6. To simulate latency, we use the King dataset[4], an end-to-end host latency dataset measured using King's method. As the King dataset contains only average latency values, we use the PingER[5] trace to simulate jitter, we use a one-day ping data set from PingER, which contains 50 pairs of end-to-end ping history for European hosts. From the 56,856 values in the PingER dataset, we create a distribution of jitter (additional latency) observed between pairs of host. Then, we use in our simulation latencies calculated as $l + \Delta l$, where l is sampled from the King dataset, and Δl is sampled from the jitter distribution.

We look at two metrics: the response time and the stall time, which we define as follows. The response time is defined as the time gap between the moment when a command is issued and the moment when it is executed/perceived on the players' computer. The stall time is the time for a client/server to wait to advance the game, the lower value the better. Each experiment is repeated 10 times, and the metrics shown are the average values.

Figure 20 (left) shows the stall time of EBLS and AoS with all the mechanisms. Dynamic delay-execution (dynamic-0 and dynamic-2) and static-max mechanisms can keep the stall time negligible, while the others cause the gaming operation to slow down significantly. Compared to AoS with dynamic delay-execution, EBLS has much higher stall time per tick, because in EBLS the tick advancement depends on the slowest clients. For all the mechanisms that exhibit high stall time, the quality of experience of players will suffer. In contrast, the result indicates that the dynamic delay-execution performs well under Internet conditions.

Figure 20 (right) shows the response time. The static-max mechanism has the highest response time, while the dynamic-2 achieves the lowest response time. EBLS has higher response time than dynamic-2 and slightly lower response time than dynamic-0. Figure 20 (right) also shows that by setting F higher (increased unfairness between players with fast and slow Internet connections), the response time is lower.

[4]https://pdos.csail.mit.edu/p2psim/kingdata/, [5]http://www-iepm.slac.stanford.edu/pinger/